HPF の性能評価と応用に関する研究

森井 宏幸，坂上 仁志，新居 学，高橋 豊
姫路工業大学　大学院工学研究科　電気系工学専攻

HPF では，ユーザがデータの分散を明示的に指示し，処理の分割，通信を行うプログラムは Owner Computes Rule に基づきコンパイラが自動的に生成する．このため，HPF を用いると従来の逐次プログラムに僅かな指示文を挿入するだけで効率のよい並列化が可能である．本稿では，共有メモリ型並列計算機システムの性能を評価するために作成された OpenMP ベンチマークプログラムである SPEC OMP2001 を HPF へと書き換え，並列性能を比較した．また，書き換えを行う上で明らかになった問題点について解決案を議論し，その問題点の解決方法を示す．

Evaluation of Parallel Performance and Programming Applicability with HPF

Hiroyuki MORII, Hitoshi SAKAGAMI, Manabu MANABU and Yutaka TAKAHASHI
Computer Engineering, Himeji Institute of Technology

Once user explicitly specifies data distribution, HPF compiler automatically divides DO loops for each processor and generates data transfer codes. Thus user can easily parallelize their programs with HPF. In this paper, we rewrite some SPEC OMP 2001 benchmark programs, which were designed to assess a shared memory type parallel computer, into HPF, and evaluate parallel performance and programming applicability. We also point out defects of HPF and discuss the solution for them.

1．はじめに

現在，実用的な並列プログラミング環境として MPI，OpenMP，HPF が存在する．MPI は，多くの異なったアーキテクチャの並列計算機で利用可能である．しかし，ユーザがプロセッサ間のデータ転送を，プログラム実行の流れを意識した上で明示的に記述しなければならず，一般ユーザには扱い難い．一方，OpenMP は一般ユーザーにも扱い易いように，従来の逐次プログラムに最小限の付加的な指示文を追加するだけでプログラムの並列実行を可能とする記述性の高い指示文ベース処理を用いているが，共有メモリ型の並列計算機でしか利用できない．

これに対して，HPF は OpenMP と同様に指示文ベース処理を用いたデータ並列言語であり，一般ユーザにも扱い易く，共有メモリ型の並列計算機だけではなく分散メモリ型の並列計算機でも利用できる．そのため，HPF は今後並列計算機を一般ユーザでも容易に扱えるツールとするために必要なプログラミング言語と言える．そこで本稿では，HPF と同様に扱い易い指示文ベースである OpenMP で記述された，共有メモリ型並列計算機システムの性能を評価するためのベンチマークプログラムである SPEC OMP2001 を HPF に書き換え，並列性能を比較した．また，書き換えを行う上で明らかになった HPF の抱える問題点について議論し，その問題点の解決方法を示す．

2．HPF による並列化

2．1　並列化の方針

本稿では，比較評価を行うために OpenMP によって並列化されている部分（並列領域）を HPF を用いて並列化する．

```
!$OMP PARALLEL DO
      do j = 1 , n
       do i = 1 , m
        u(i,j) = p(i,j)
        v(i,j) = p(i,j+1)
       enddo
      enddo
!$OMP END PARALLEL
```

上記のような並列領域があるとき，DO 変数 j の DO ループが並列化されているため，HPF で

はDOループ内の変数u,v,pを以下のようにjの次元である第2次元目でデータ分散する．

```
!HPF$ PROCESSORS proc(n)
!HPF$ DISTRIBUTE (*,BLOCK) ONTO
!HPF$& proc :: u,v,p
```

また，式の中に異なる添字を持つ変数があるとき，DISTRIBUTE指示文によりデータ分散を行っただけでは，特定のループ繰り返し中の計算に必要なデータが異なるプロセッサに分散されている場合があり，このままでは効率よく通信が行われない．そこで，シャドウ領域を確保し，REFLECT指示文を用いることでブロック通信を行う．上記の例では，vの代入文でpの2次元目の添字がj+1であるため，それに見合ったシャドウ領域が必要となる．このため，以下のようにシャドウ領域を確保する．

```
!HPF$ SHADOW (0,0:1) :: p
```

宣言部でこれまでの指示文の挿入する．また，INDEPENDENT指示文を用いることでDOループが並列実行可能であることを示し，REFLECT指示文を用いてシャドウ実体を更新し，以下のようにON-HOME-LOCAL指示文により並列実行には通信の必要がないことを明示する．

```
!HPFJ REFLECT p
!HPF$ INDEPENDENT
      do j = 1 , n
!HPFJ ON HOME(u(i,j)), LOCAL BEGIN
       do i = 1 , m
        u(i,j) = p(i,j)
        v(i,j) = p(i,j+1)
       enddo
!HPFJ END ON
      enddo
```

基本的には以上のような指示文の挿入により，OpenMPで記述されたプログラムをHPFに書き換える．

なお，実行環境には大阪大学サイバーメディアセンターのNEC SX-5を用いた．

2．2　SWIM

SWIMは差分法により浅水方程式を数値計算し，海水のシミュレーションを行うプログラムである．

SWIMの書き換えでは，同一のDOループ内で結果を格納する複数の変数の添字が一致していない場合に並列化を行うと効率よく実行が行われない問題が見つかった．この問題を不適切分散問題と呼ぶ．

```
      dimension
     & u(10,10),v(10,10),p(10,10)
!HPF$ PROCESSORS proc(2)
!HPF$ DISTRIBUTE (*,BLOCK) ONTO proc
!HPF$&  :: u,v,p
      ･･･
!HPF$ INDEPENDENT
      do j = 1 , 9
       do i = 1 , 9
        u(i+1,j) = -p(i+1,j+1)
        v(i,j+1) =  p(i+1,j+1)
       enddo
      enddo
      ･･･
```

上記の例では，u(i,1:5)がproc(1)にu(i,6:10)がproc(2)にデータ分散されている．同様にv，pもデータ分散されているため，proc(1)はu (i,5)を所持しているが，v(i,6)は所持していない．このため，j=5のループ処理は，Owner Computes Ruleにより，異なるプロセッサで行わなければならない．しかし，NEC SX-5のHPFコンパイラはDOループ処理の分割を行うとき，DOループの範囲を一塊として扱うため，Owner Computes Ruleとの間に矛盾が生じる．

この矛盾を解決するために一時配列vtmpが動的に確保され，vtmp(i.j)=p(i+1,j+1) と結果が格納される．そして，演算後vtmp(i,j) がv(i,j+1) にコピーされる．この配列の動的確保，解放と配列間のコピーのため並列実行の効率が落ちていた．

不適切分散問題は，同一ループ内で結果を格納する複数の変数の添字が一致していない場合に生じるため，以下に示すようにDOループを分割することで問題を回避できる．

```
      ･･･
!HPF$ INDEPENDENT
      do j = 1 , 9
       do i = 1 , 9
        u(i+1,j) = -p(i+1,j+1)
       enddo
```

```
      enddo
!HPF$ INDEPENDENT
      do j = 1 , 9
       do i = 1 , 9
        v(i,j+1) =  p(i+1,j+1)
       enddo
      enddo
      •••
```

また，以下に示すようにALIGN指示文を用いて結果を格納する変数の添字に合わせてデータ分散を行うことによっても不適切分散問題を解決できる．

```
      dimension u(10,10)
     & ,v(10,10),p(10,10)
!HPF$ PROCESSORS proc(2)
!HPF$ DISTRIBUTE (*,BLOCK) ONTO proc
     & :: u,p
!HPF$ TEMPLATE , DISTRIBUTE(*,BLOCK)
     & ONTO proc :: base(10,0:10)
!HPF$ ALIGN v(i,j) WITH base(i,j-1)
      •••
!HPF$ INDEPENDENT
      do j = 1 , 9
       do i = 1 , 9
        u(i+1,j) = -p(i+1,j+1)
        v(i,j+1) =  p(i+1,j+1)
       enddo
      enddo
      •••
```

DOループ分割を用いると，簡単に不適切分散問題を解決することができる．しかし，DOループ内でベクトルレジスタのデータが再利用できる場合，DOループを分割するとデータのリロードが必要になり，ベクトル処理の効率が悪くなる．このため，DOループ分割による不適切分散問題の解決は，計算効率の悪化を伴う場合がある．一方，ALIGN指示文を用いる方法も，適切なデータ分散がDOループによって異なる場合，すべてのDOループに一つのALIGN指示文で対応できないという問題がある．

次の例では，上のDOループはALIGN指示文を用いてu(i+1,j)とv(i,j+1)を同じプロセッサに分散することで不適切分散問題を解決することができる．しかし，そのようにデータ分散を行うと下のDOループでj=6のループのとき不適切分散問題が生じてしまう．

```
      •••
      do j = 1 , 9
       do i = 1 , 9
        u(i+1,j) = -p(i+1,j+1)
        v(i,j+1) =  p(i+1,j+1)
       enddo
      enddo
      •••
      do j = 1 , 9
       do i = 1 , 9
        u(i,j) = x(i,j)
        v(i,j) = y(i,j)
       enddo
      enddo
      •••
```

この場合，上のDOループを分割することで不適切分散問題を解決する方法と上のDOループはALIGN指示文を用いて不適切分散問題を解決し，下のDOループは分割することで新たな不適切分散問題が生じないようにする方法が考えられる．この二つの方法を比較すると，後者の方がベクトルレジスタのデータを再利用できて効率がよいため以下のように並列化を行う．

```
!HPF$ DISTRIBUTE (*,BLOCK)
     & ONTO proc :: u
!HPF$ TEMPLATE , DISTRIBUTE(*,BLOCK)
     & ONTO proc :: base(10,0:10)
!HPF$ ALIGN base(i,j) WITH v(i,j-1)
      •••
!HPF$ INDEPENDENT
      do j = 1 , 9
       do i = 1 , 9
        u(i+1,j) = -p(i+1,j+1)
        v(i,j+1) =  p(i+1,j+1)
       enddo
      enddo
      •••
      do j = 1 , 9
       do i = 1 , 9
        u(i,j) = x(i,j)
       enddo
      enddo
      do j = 1 , 9
       do i = 1 , 9
        v(i,j) = y(i,j)
       enddo
      enddo
      •••
```

このように不適切分散問題は，主にALIGN指示文を用いて解決し，ALIGN指示文のみで対応

できない場合は効率を考慮にいれてDOループの分割を併用することで解決した．以上の方法によりHPF並列化したプログラムの実行時間と高速化率を表１に示す．

表１ SWIMの計測結果

		実行時間[秒]			高速化率		
		改善前	改善後	OpenMP	改善前	改善後	OpenMP
プロセッサ数	1	505.3	294.7	283.2	1.00	1.00	1.00
	2	265.9	153.3	150.4	1.90	1.92	1.88
	3	183.7	109.2	103.2	2.75	2.70	2.74
	4	144.4	82.3	81.4	3.50	3.58	3.48
	8	81.8	51.3	48.4	6.18	5.74	5.85

表1の結果より，不適切分散問題を解決することで性能を大幅に改善することができ，HPFでOpenMPと同等の並列性能が実現できた．

2．3　MGRID

MGRIDはマルチグリッドで3次元のポテンシャル場を計算するプログラムである．

MGRIDでは，以下のように大きな1次元配列を主プログラムで用意し，副プログラムではその1次元配列の一部を3次元配列として用いている．

```
      dimension a(M),is(k),il(k)
      •••
      do i = 1 , k
        call f(a(is(i)),il(i))
      enddo
      •••
      stop
      end

      subroutine f(a,m)
      dimension a(m,m,m)
      •••（計算）
      return
      end
```

引数として渡す配列変数の次元数が呼ぶ側と呼ばれる側のプログラムで異なる，いわゆる順序結合はFORTRAN77を用いて大規模なプログラムを作成するときにはよく用いられた．しかし，HPFは並列化を行うためにデータ分散を行うが，分散配列の順序結合には対応していない．そのため，MGRIDのプログラムはHPFの指示文を追加するだけでは並列化できなかった．HPFが分散配列の順序結合をサポートしていない問題を引数の次元数不一致問題と呼ぶ．

呼ぶ側のプログラムの次元数に合わせて呼び出されたプログラム側にデータを渡し，呼び出されたプログラム側で実際に扱う次元数で必要なサイズの配列を動的に用意し，渡されたデータをコピーして使用することで引数の次元数不一致問題を回避した．副プログラムで扱う配列のサイズが呼ぶタイミングで異なるために副プログラムで用意する配列は動的に確保するのが妥当である．なぜなら，配列は必要以上に確保すると処理の分割が均等にならないためである．以下にMGRIDで引数の次元数不一致問題を動的配列へのコピーし解決した例を簡略化して示す．

```
      dimension a(M),is(k),il(k)
      •••
      do i = 1 , k
        call f(a(is(i)),il(i))
      enddo
      •••
      stop
      end

      subroutine f(aa,m)
      dimension aa(m**3)
      allocatable a(:,:,:)
!HPF$ PROCESSORS proc(N)
!HPF$ DISTRIBUTE (*,*,BLOCK)
     & ONTO proc :: a
      allocate a(m,m,m)
      a ← aa にコピー
      •••（計算）
      aa ← a にコピー
      deallocate(a)
      return
      end
```

主プログラム側から部分配列として変数配列aの一部をm**3のサイズで副プログラムfにaaとして渡し，整合配列を用いてデータを副プログラム側で受け，本来の３次元配列aにデータをコピーするように書き換えている．以上のソースコード修正を含むHPFプログラムの実行時間と高速化率を表２に示す．

表２ MGIRDの計測結果

		実行時間[秒]		高速化率	
		HPF	OpenMP	HPF	OpenMP
プロセッサ数	1	285.3	230.2	1.00	1.00
	2	216.2	118.6	1.32	1.94
	3	176.1	81.1	1.62	2.84
	4	160.9	63.1	1.77	3.65
	8	138.8	36.0	2.06	6.39

動的配列の確保，解放とコピーのために，OpenMPほどの性能は得られなかった．今後の課題としてよりよい解決法の検討，簡単なソース書き換えにより問題を回避する方法の検討があげられる．

２．４ APSI

APSIは，3次元流体を用いて数値計算を行い，湖の環境における汚染物質の拡散をシミュレーションするプログラムである．

APSIでは，以下のように大きな1次元配列を確保し，副プログラムではその一部を配列変数として用いていた．また，引数の次元数不一致問題となる部分も存在した．

```
      allocatable :: work(:)
      •••
      l3 = nx*ny*nz
      lc=1
      last = last + l3
      lstepc = last
      •••
      call run(nx,ny,nz,work(lc),
     & work(lstepc))
      •••

      subroutine run(nx,ny,nz,c,
     & stepc)
      dimension c(*),stepc(*)
      •••
      call dctdx(nx,ny,nz,c)
      •••

      subroutine dctdx(nx,ny,nz,c)
      dimension c(nx,ny,nz)
```

MGRIDでは，同じ変数名を異なるサイズの変数配列が利用していたので動的に配列を確保する手法をとった．しかし，動的に確保，解放，コピーを行うことは効率が悪かった．一方，APSIでは変数のサイズが変更されることがないため副プログラムで利用する配列変数と同じ形式で定義を行った．

また，その配列変数が異なった次元数で定義される場合，次元数の多い方の次元数で定義を行った．そして，その変更に合わせてプログラムの書き換えを以下のように行うことで並列化を行った．

```
      allocatable ::
     & c(:,:,:),stepc(:,:,:)
      •••
      allocate(c(nx,ny,nz))
      allocate(stepc(nx,ny,nz))
      •••
      call run(nx,ny,nz,c,stepc)

      subroutine run(nx,ny,nz,c,
     & stepc)
      dimension c(nx,ny,nz),
     & stepc(nx,ny,nz)
      •••
      call dctdx(nx,ny,nz,c)
      •••

      subroutine dctdx(nx,ny,nz,c)
      dimension c(nx,ny,nz)
```

APSIでは並列化を行う次元が並列化されている場所によって異なる場合がある．HPFではデータ分散を行っている次元でしか処理分割できないので，並列化に適した次元で適宜データを再分散しなければならない．このデータ再分散は，以下に示すように副プログラム呼び出し時の引数再マッピングで行った．

```
      •••
!HPF$ DISTRIBUTE (*,*,BLOCK) ONTO P
     & :: c
      ••• （３次元目で分散された計算）
      call advc(nx,ny,nz,c)
      •••

      subroutine  advc(nx,ny,nz,c)
      •••
!HPF$ DISTRIBUTE (*,BLOCK,*) ONTO P
     & :: c
      ••• （2次元目で分散された計算）
```

また，以下のように副プログラムが並列に呼び出されるときにiの値に応じてcのnxny個のデータが引数として渡されている場合があった．

```
      dimension c(nx,ny,nz)
      •••
      mlag = 1
!$OMP PARALLEL DO
!$OMP&PRIVATE(mlag)
      do i = 1 , nz
        mlag = mlag + nxny
        call f(nx,ny,c(mlag))
      enddo
!$OMP END DO
!$OMP END PARALLEL
      •••
      subroutine f(nx,ny,c)
      dimension c(nx,ny)
      •••
```

HPFでは，分散された配列をそのまま並列呼び出しを行う副プログラムへは渡せないため，以下に示すように必要なデータのみを渡す形状引継配列を用いて書き換えた．

```
      dimension c(nx,ny,nz)
!HPF$ DISTRIBUTE (*,*,BLOCK) ONTO P
     & :: c
      •••
!HPF$ INDEPEDENT
      do i = 1 , nz
        call f(nx,ny,c(:,:,i))
      enddo
      •••
      subroutine f(nx,ny,c)
      dimension c(nx,ny)
      •••
```

以上により並列化されたHPFプロログラムの実行時間と高速化率を表3に示す．

表3　APSIの計測結果

		実行時間[秒]		高速化率	
		HPF	OpenMP	HPF	OpenMP
プロセッサ数	1	1495.9	928.4	1.00	1.00
	2	763.2	477.6	1.96	1.94
	3	524.3	311.6	2.83	2.98
	4	388.8	235.4	3.84	3.94
	8	271.0	136.6	5.52	6.80

並列化を行う次元が異なるために再マッピングを行ったため，再マッピングによるデータ転送などが負担となりOpenMPほどの性能は出せなかった．しかし，高速化率を引き出すことはできた．

3. まとめ

SPEC OMP2001ベンチマーク中のプログラムSWIM，MGRID，APSIをHPFで書き換え，大阪大学サイバーメディアセンターのNEC SX-5を用いて実行時間と高速化率を計測した．また，その中で明らかになった問題点の解決に取組み性能評価を行った．

SWIMの書き換えでは，不適切分散問題が見つかった．この問題は，ALIGN指示文の利用とDOループの分割を行うことで解決できることが解った．MGRIDの書き換えでは，引数の次元数不一致問題が見つかった．この問題自体は，一度必要な次元数の配列にコピーすることで回避できたが高い高速化率を得られなかった．

APSIでは，並列化を行う次元が異なる場合が存在した．この問題は副プログラム毎に適したデータ分散を指定して副プログラム呼び出し時に再マッピングを行うことで，OpenMPと同様に異なる次元での並列化を実現した．

HPFは不適切分散問題，引数の次元数不一致問題のように十分な対応ができていない部分もあるが，工夫することで並列化が可能であることが解った．

謝辞

本研究の実施において，並列計算機NEC SX-5の利用環境を提供いただいた大阪大学サイバーメディアセンターに謝意を表する．

参考文献

[1] High Performance Fortran2.0 公式マニュアル，High Performance Fortran Forum，1999

[2] HPFプログラミングガイド，地球シュミレータ技術開発センター，2002